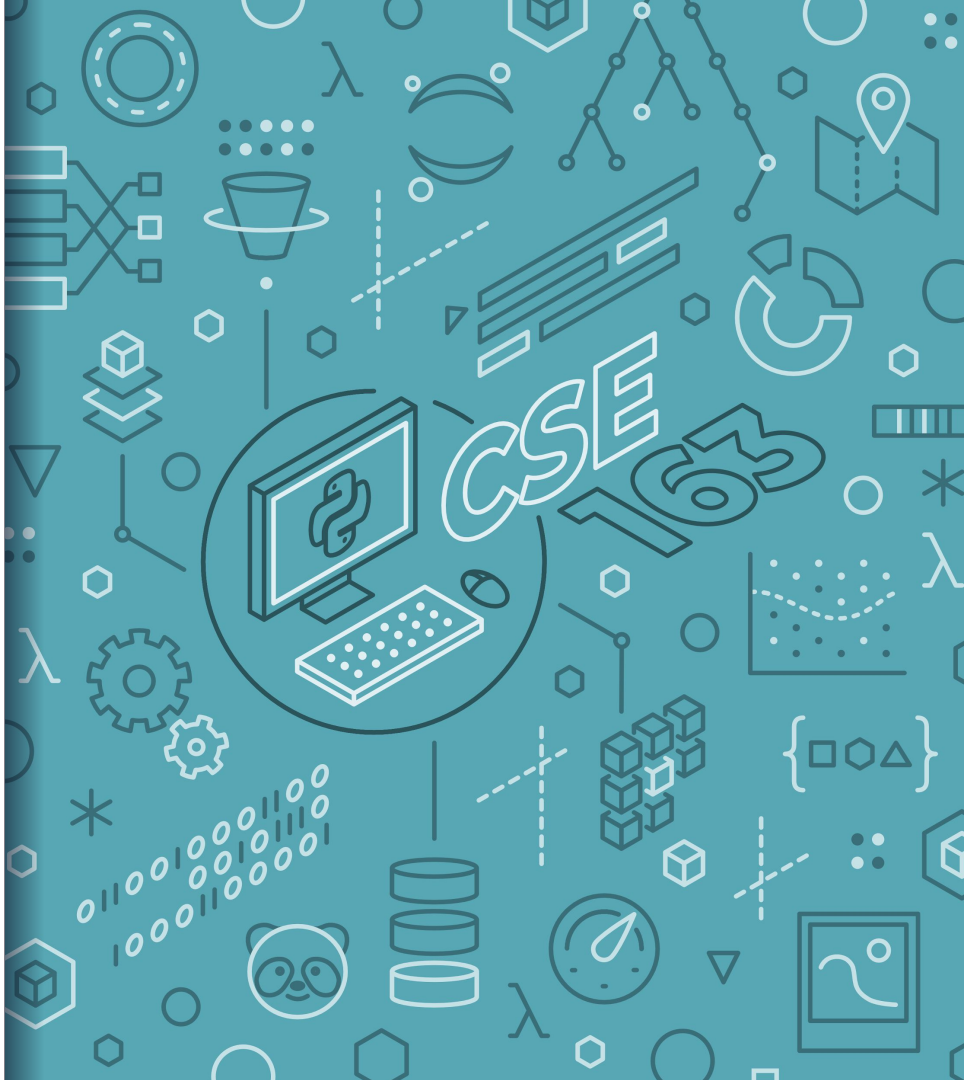
CSE 163

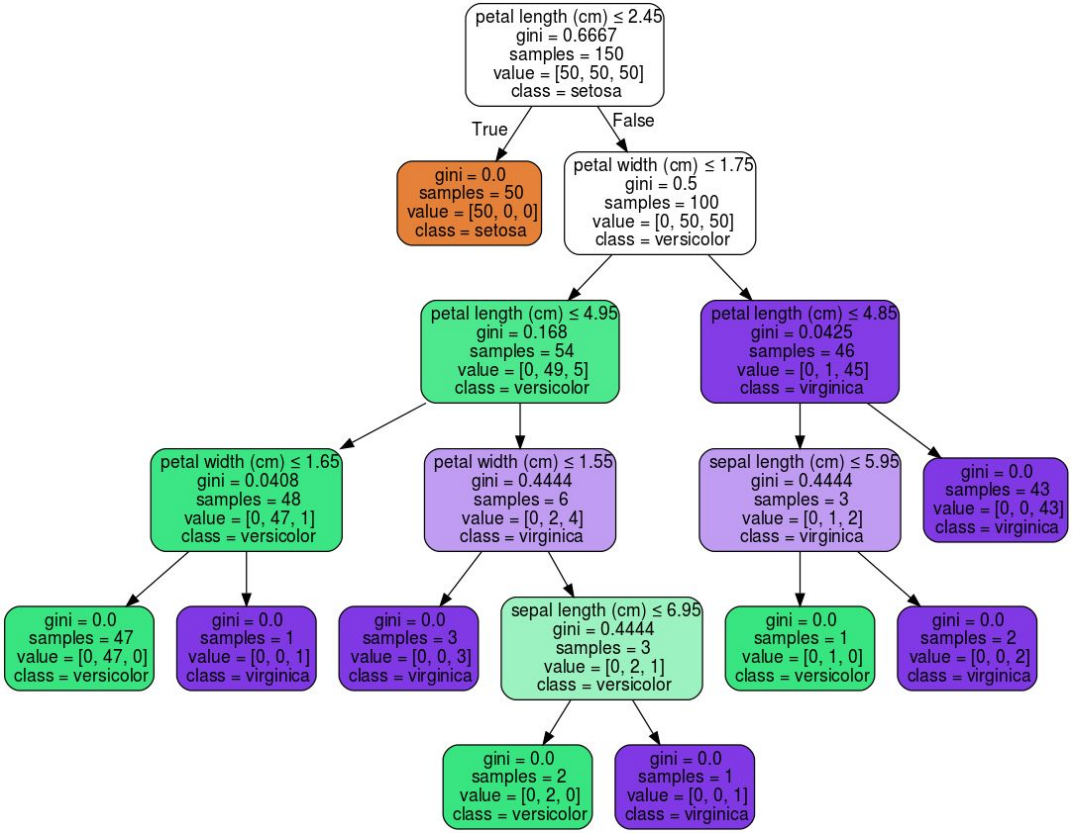
Classes and Objects

Hunter Schafer



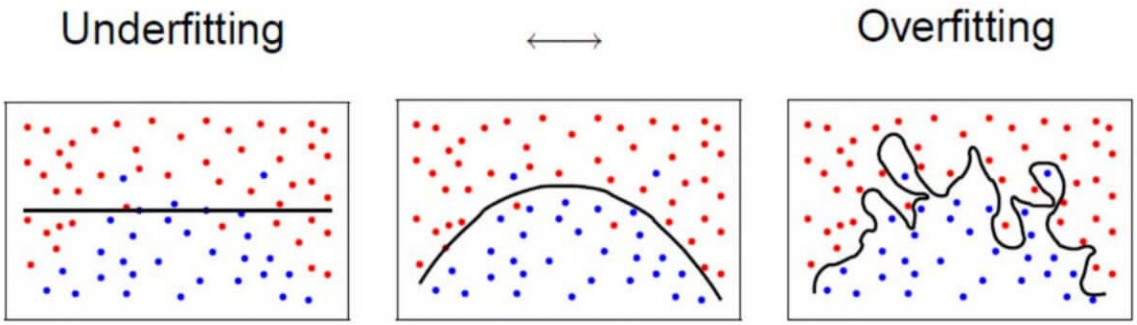
Last time on...

Decision Tree



Overfitting

- The most important problem in science you've never heard of
- Overfitting: When your model matches the training set so well, that it fails to generalize
  - Memorizing answers to Multiple Choice test
- Tall trees are likely to overfit if you don't have enough data
  - Can learn very complex boundaries
  - Very few points at the leaves



Evaluating Models

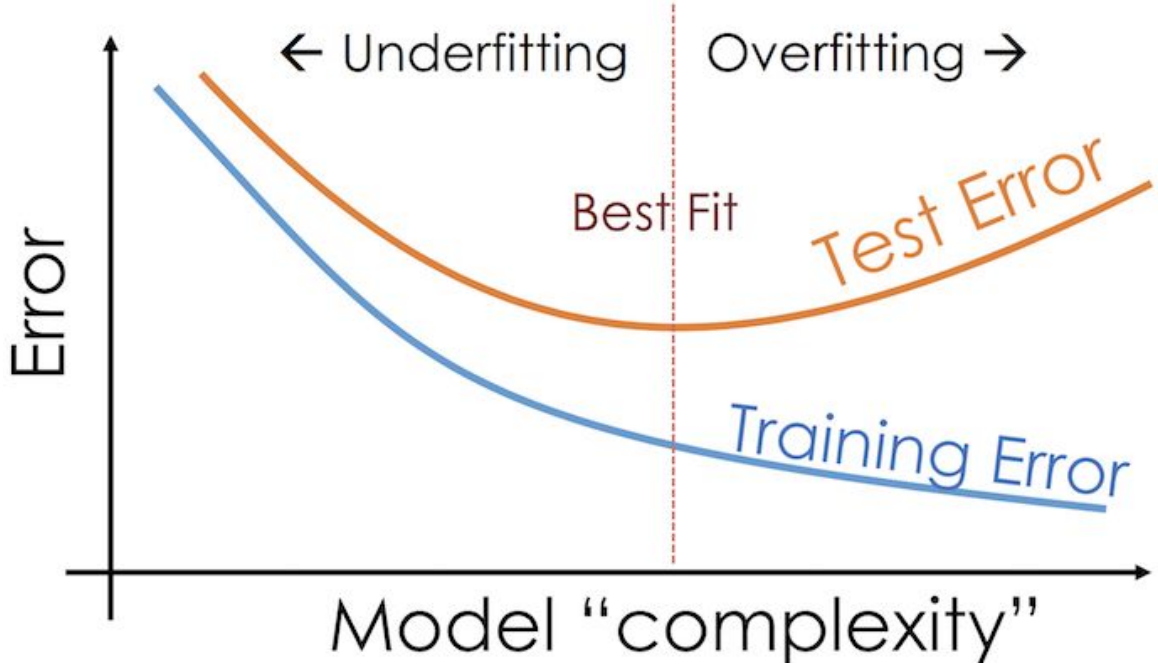
- Training is cool, but we want to know its future performance
- Training data can't be used to evaluate model
  - "I got 100% on the practice test I have been studying for 4 hours, therefore I will get 100% on the exam"
- Must hold out data called a **test set** to evaluate at the end
  - Unbiased estimate of performance in the wild

Never ever ever train or make decisions based on your test set.

If you do, it will no longer be good estimate of future performance.

Model Complexity

Note this is error, not accuracy (`error = 1 - accuracy`)

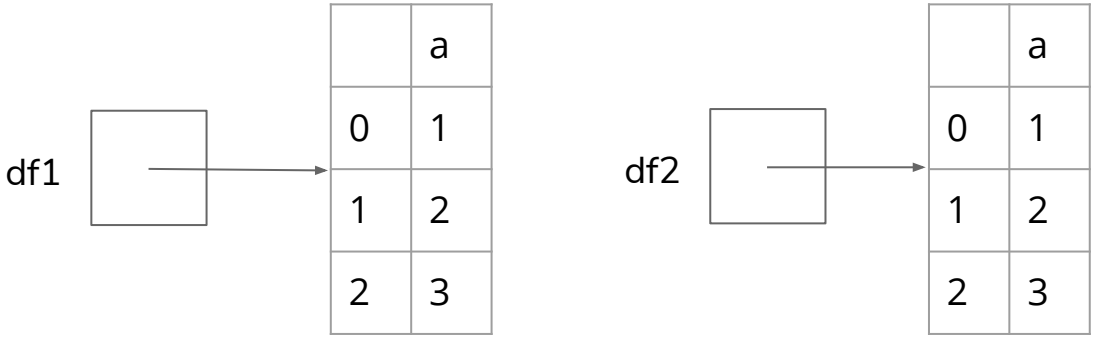


This week!

Objects

- An **object** holds state and provides behaviors that operates on that state
- Each object has its own state
- Example: DataFrame

```
df1 = pd.DataFrame({'a': [1, 2, 3]})
df2 = pd.DataFrame({'a': [1, 2, 3]})
```



DataFrame

- State
  - The columns
  - The index
  - The actual data
  - Etc.
- Behaviors
  - Methods for providing access to the data
  - Methods to modify data
  - Methods to find/replace missing values
  - Etc.

Class

- A **class** lets you define a new object type by specifying what state and behaviors it has
- A class is a blueprint that we use to construct **instances** of the object

Here is a full class

```
class Dog:
  def __init__(self, name):
    self.name = name

  def bark(self):
    print(self.name + ': Woof')
```

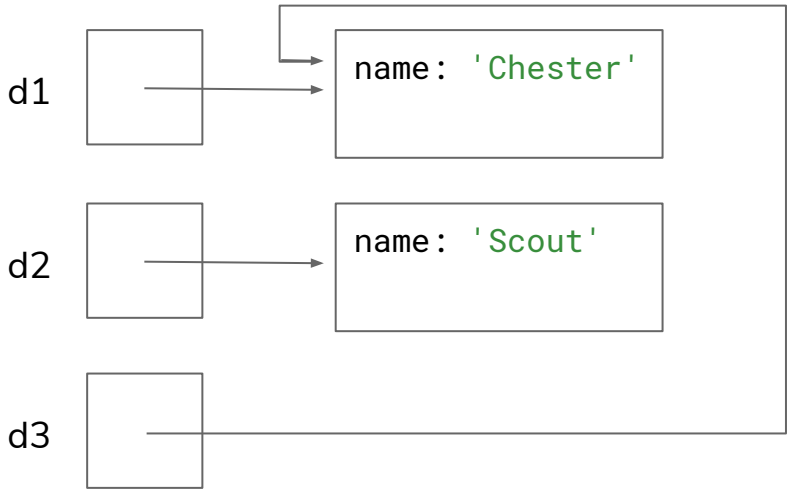
A class definition

An initializer that sets fields **(state)**

A method **(behavior)**

Building Dogs

```
d1 = Dog('Chester')
d2 = Dog('Scout')
d3 = d1
d1.bark()  # Chester: Woof
d2.bark()  # Scout: Woof
d3.bark()  # Chester: Woof
```



Note to Self

- In the Dog example, every method (initializer or otherwise) takes a parameter called “self”
- This indicates which **instance** the method is being called on

```
class Dog:
  def __init__(self, name):
    self.name = name

  def bark(self):
    print(self.name + ': Woof')
```

```
d1.bark()
d2.bark()
d3.bark()
```

- On the first line, self references Chester
- On the second line, self references Scout
- On the third line, self references Chester

Coding with Classes

VS Code

- Can split program into multiple files
- Put the Dog class in dog.py

```
# Syntax 1
import dog
d = dog.Dog('Chester')

# Syntax 2
from dog import Dog
d = Dog('Chester')
```

Poll Everywhere

Think

1 minute

pollev.com/cse163

For this program, draw the memory model for the objects and then select which option best represents your model.

```
d1 = Dog('Chester')
d2 = Dog('Scout')
d3 = d1

p1 = DogPack()
p1.add_dog(d1)
p1.add_dog(d2)

p2 = DogPack()
p2.add_dog(d3)
```

Poll Everywhere

Pair

2 minutes

pollev.com/cse163

For this program, draw the memory model for the objects and then select which option best represents your model.

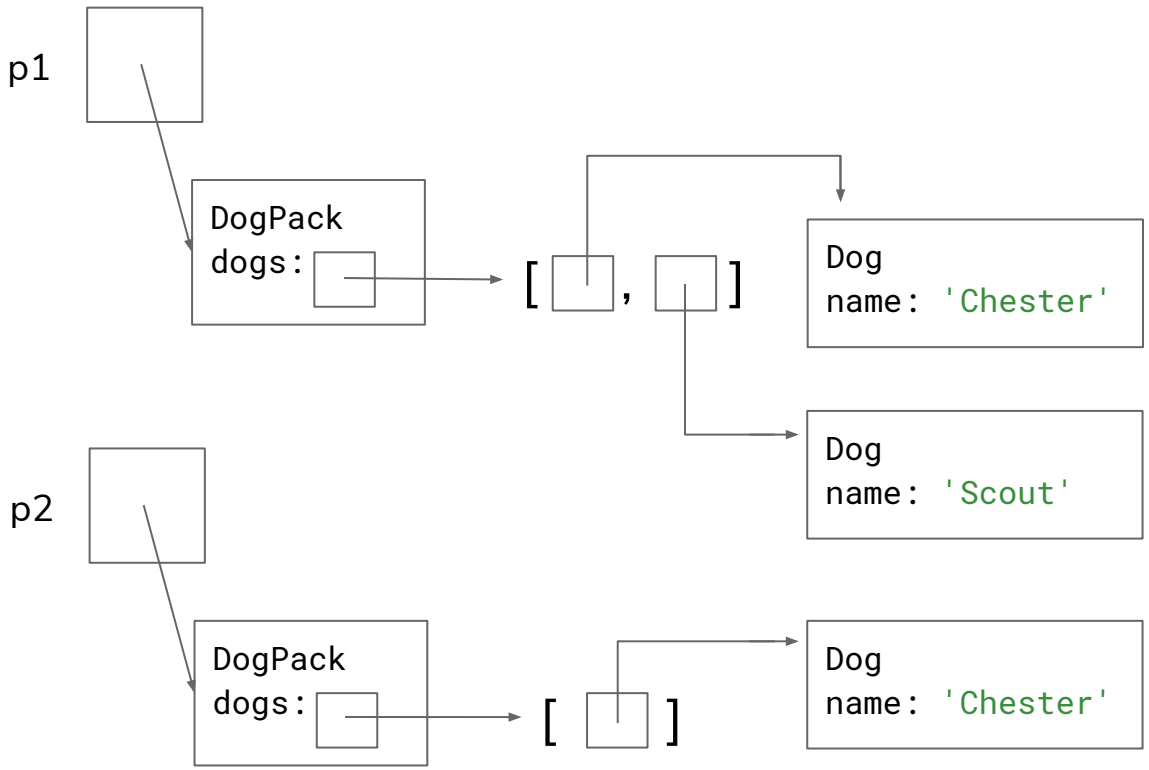
```
d1 = Dog('Chester')
d2 = Dog('Scout')
d3 = d1

p1 = DogPack()
p1.add_dog(d1)
p1.add_dog(d2)

p2 = DogPack()
p2.add_dog(d3)
```

Poll Everywhere

Think

Not a real slide

pollev.com/cse163

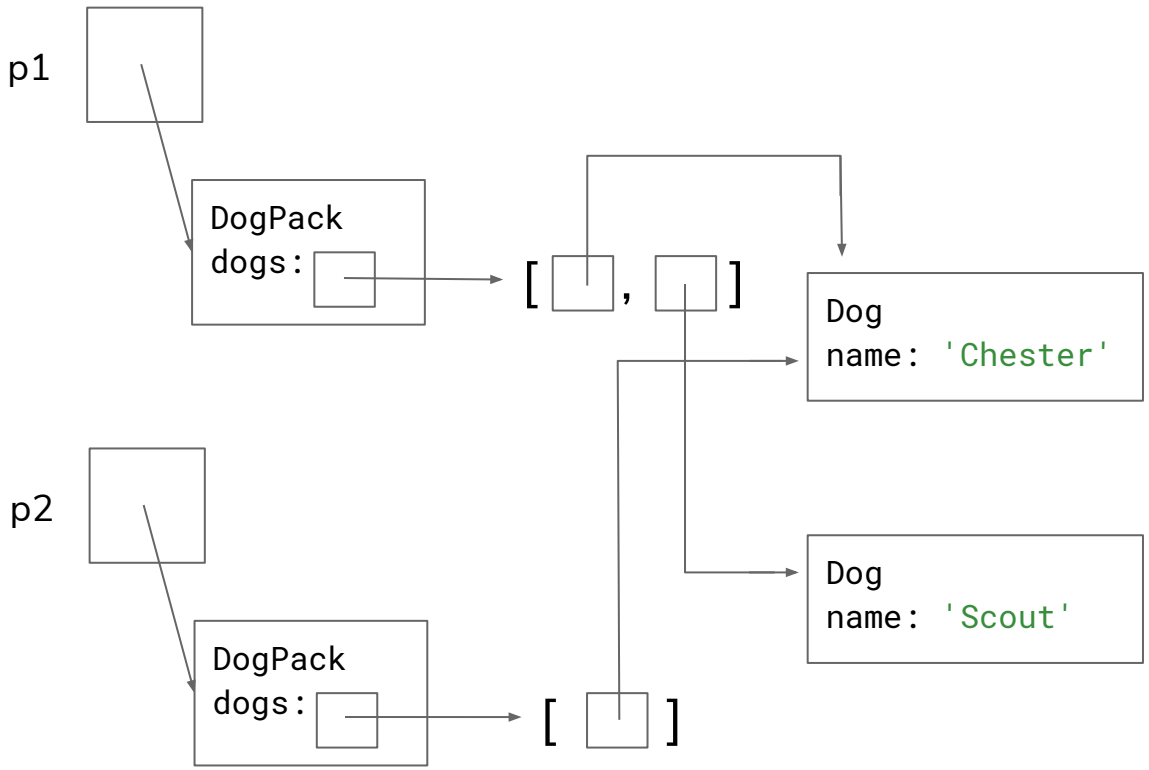


Poll Everywhere

Think

Not a real slide

pollev.com/cse163



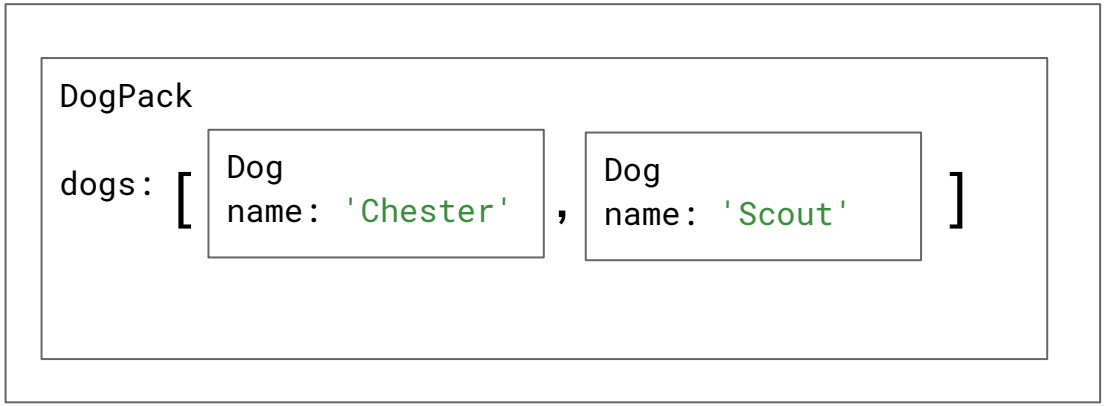
Poll Everywhere

Think

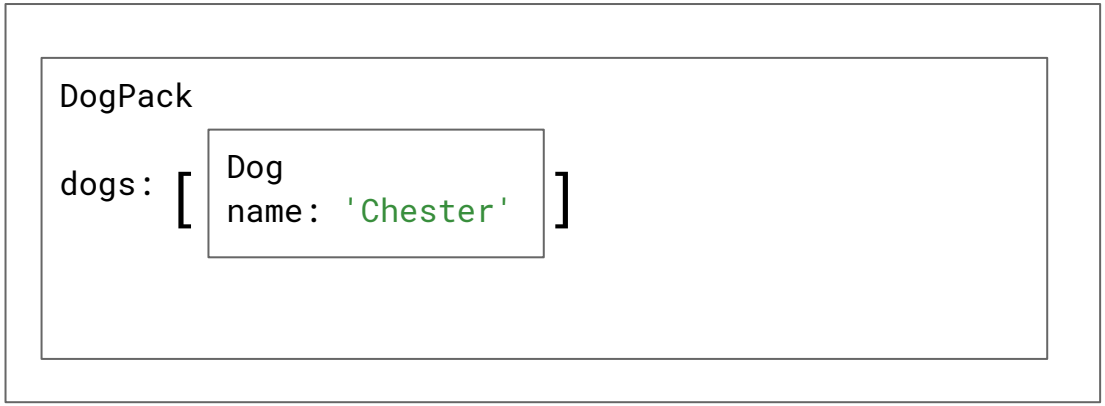
Nota real slide

pollev.com/cse163

p1



p2



Private

- Objects are great, but it's not desirable if the client can change our state without us "knowing" about it

```
p = DockPack()
p.add_dog(Dog('Chester'))

p.dogs = None

p.add_dog(Dog('Scoute'))  # Crashes
```

- It would be nice if we could restrict the client to just using the behaviors we provide

Private

- Python has no way to actually do this, but by convention people don't access things that start with "_"

```
class DogPack:
    def __init__(self):
        self._dogs = []

    def add_dog(self, dog):
        self._dogs.append(dog)

    def _private_method(self):
        print('Some helper method')
```